UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Analyzing the security implications of the CXL.cache protocol

Analyse der Sicherheitsrisiken des CXL.cache Protokolls

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Niklas Dörfling

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Thore Tiemann

Lübeck, den 09. Dezember 2025

Zusammenfassung

Obwohl Compute Express Link (CXL) kürzlich viel Aufmerksamkeit für dessen Teilprotokoll CXL.mem bekam, wurde das Teilprotokoll CXL.cache bisher wenig untersucht. Wir untersuchen die Sicherheitsrisiken bei Nutzung von CXL.cache fähiger Hardware in Cloud-Umgebungen. Wir fokussieren uns dabei auf mikroarchitekturelle Angriffe für den Fall, dass sich mehrere Kunden dieselbe Hardware teilen.
Um das zu tun, untersuchen wir das Protokoll praktisch mit einem Intel FPGA basierten Beschleuniger und finden heraus, dass ein CXL Gerät durch Zeitmessungen präzise ermitteln kann ob und wo Daten zwischengespeichert sind. Wenn mehrere Lesezugriffe getätigt werden, kann sogar ermittelt werden, ob die Daten modifiziert vorliegen im Cache. Außerdem zeigen wir, dass CXL.cache Zugriffe Daten direkt in den LLC des Hosts schreiben können.
Am wichtigsten ist, dass wir hervorheben, dass CXL.cache Zugriffe physische Speicheradressen benutzen ohne jegliche Interaktion mit der IOMMU, was bedeutet, dass Speicherisolation nicht über die IOMMU oder ähnliche Mechanismen geregelt werden kann. Das macht unser angenommenes Szenario der geteilten Hardware unrealistisch, weil ein Angreifer mit vollem Zugriff auf physischen Speicher keine mikroarchitekturellen Angriffe nutzen muss, um sensible Daten zu klauen.
Zuletzt untersuchen wir kurz, ob CXL.cache benutzt werden kann, um Intel TDX Enklaven zu attackieren und skizzieren wie Speicherisolation mit CXL.cache vereinbart werden könnte.

Abstract

While CXL has recently gained attention for its subprotocol CXL.mem, the subprotocol CXL.cache has mostly been overlooked. We assess the implications for security when using CXL.cache enabled hardware in cloud environments, with a focus on micro-architectural attacks in multi-tenant scenarios.

For that, we investigate the protocol using an Intel FPGA based accelerator and find that a CXL device can accurately determine if and where a line of memory is cached by measuring access latencies. When issuing multiple read requests, it is also possible to distinguish between clean and dirty cache lines. Moreover, we show that CXL.cache requests can directly push cache lines into the LLC of the host.

Most importantly, we highlight that CXL.cache requests operate on physical addresses and naturally bypass the IOMMU, meaning that memory isolation cannot be enforced using the IOMMU or similar mechanisms. This renders our initially assumed multi-tenant scenario obsolete, since an attacker with full physical memory access would not need to use micro-architectural attacks to exfiltrate sensitive data.

Lastly, we briefly examine whether CXL.cache can be leveraged against Intel TDX enclaves and outline how memory isolation could be integrated into CXL.cache.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 09. Dezember 2025

Danksagungen

Zu erst möchte ich mich bei dem CTF Team, besonders Anja, Jonah und Thore, bedanken. Ihr habt mein Studium maßgeblich geprägt. Ich denke noch heute gerne an das erste Turnier in der Weihnachtszeit 2022, an dem ich mit euch teilgenommen habe.
Außerdem bin ich sehr dankbar dafür, dich als meinen Betreuer zu haben, Thore. Nicht nur haben unsere wöchentlichen Treffen mir sehr geholfen, meine Fragen schnell zu klären und neue Inspirationen zu bekommen, du warst auch immer schnell zur Hilfe, wenn der Server nach einem Experiment mal wieder abgestürzt ist. Vielen Dank für alles, was ich die letzten Jahre von dir lernen durfte.
Ebenfalls möchte ich mich bei meiner Familie bedanken, die in jeglichen Aspekten immer für mich dagewesen ist. Vielen Dank auch dafür, dass ihr mich so sehr bei meinem Umzug unterstützt habt. Ich danke auch dir, Laura, dass du diese Arbeit korrekturgelesen hast und mich in der zuletzt stressigen Zeit dieser Arbeit immer so herzlich aufgemuntert hast.

Contents

1 Introduction

The demand for computational power is greater than ever before, particularly driven by advancements in artificial intelligence (AI). To meet these computational demands, data centers incorporate a heterogeneous computing model and utilize special accelerators to speed up certain tasks. A classic example for such an accelerator is a graphics processing unit (GPU), which is designed for highly parallel tasks like training AI models. Alternatively, accelerators can be built using reconfigurable hardware such as field programmable gate arrays (FPGAs). Because FPGAs can be reconfigured at runtime, they are suitable for a wide range of workloads. This flexibility has driven their popularity and resulted in their adoption in cloud platforms [Ama25, Mic25b, Ali25].
To maximize their profits, cloud service providers are logically separating the compute resources of one machine across multiple tenants. As no trust between tenants is assumed, a strict separation and access control mechanisms are of critical importance. However, as the tenants are still physically on the same machine, they might be able to infer knowledge about the activity of co-located tenants. This can be achieved using micro-architectural side-channel attacks (SCAs) which monitor the micro-architectural state of the central processing unit (CPU) in order to detect activity of third parties [RTSS09, YF14].
A well-studied class of SCAs are cache attacks [RFKG25, YF14, OST06] which can be performed by interacting with the caching subsystem of the CPU. Secondary devices, like FPGAs, are often connected with the CPU in a non-coherent manner, meaning that they cannot interact with the host caches. However, the upcoming cache-coherent interconnect protocol Compute Express Link (CXL) enables devices to interact with the host caches. Therefore, it is important to study whether CXL devices are able to perform cache attacks against co-located tenants.

1.1 Scope

This work aims to gain insights into the security implications of adding CXL enabled devices to public cloud platforms. Since we are interested in potential cache side-channel attacks, we limit our research to the sub-protocol CXL.cache which enables the cache-coherent communication. We aim to exercise the protocol in practice using a lab setup consisting of an Intel Xeon 5th Gen CPU and a compatible Intel Agilex 7 FPGA board, both suitable for CXL 1.1. Implementing a CXL controller is out of scope for this work,

therefore we utilize an existing implementation of the protocol provided by Intel in form of a CXL IP core, whereas IP stands for Intellectual Property since someone else designed it. The following questions shall be answered throughout the thesis:

- Will an attacker benefit from interacting with the cache-coherency protocol compared to CPU-only attackers? If so, to what extent?
- Does the CXL IP implement all requests listed in the specification and does it limit an attacker to realize advanced attacks?

1.2 Contributions

We reproduce results from related work [PTV22], showing that a fraction of findings are applicable to CXL in Section 6.3.2 and 6.4. Moreover, we circumvent the IP encryption used by an electronic design automation (EDA) tool in Section 4.2.2, which enables us to gain further insights into the CXL.cache IP. Details on the circumvention of the IP encryption are left out since we are still in a responsible disclosure process with Intel. Our main contributions are:

- We explore direct interaction with the host last level cache (LLC) and find indications for a new Intel Data Direct I/O Technology (DDIO) related behavior in Section 6.4.
- We show that all CXL.cache requests bypass the Input-Output Memory Management Unit (IOMMU) by nature in Section 6.1 and thus the protocol does not allow for memory isolation. Due to that, the protocol cannot be securely used in a multi-tenancy scenario, as one tenant can directly compromise other tenants. Future work should therefore consider a different attacker model when assessing further security implications.
- We find that an attacker can monitor cache lines by observing snoop requests issued by the host in Section 6.2. This allows distinguishing different types of host requests, e.g., read or write requests.

1.3 Structure

We first provide required background and related work in Chapter 2. This covers computer architecture fundamentals on cache architecture and cache side-channel attacks. Furthermore, we introduce the CXL protocol fundamentals and outline the existing request types. Finally we go over Intel DDIO and related work.

The next Chapter 3 presents our experimentation setup, explaining which devices and IP blocks we use to run our experiments.
We follow up with the problems we encountered working with the CXL IP and address how we solved them in Chapter 4. This includes reverse engineering the EDA tool used to program Intel FPGAs, Quartus Prime, to circumvent its IP encryption.
Next, we document details regarding the interface to the CXL IP in Chapter 5. For instance, we outline the used bus protocols to communicate with the IP and how user requests map to raw CXL.cache requests in Section 5.1.3.
Then we present our findings in Chapter 6 and end with a conclusion and potential of future work in Chapter 7.

2 Background and Related Work

This chapter provides required background. We introduce terms commonly used when working with Intel FPGAs, explain basics on caches and how they are used in micro-architectural SCAs. Furthermore, we cover coherency protocols and introduce CXL along with related work on heterogeneous cache attacks.

2.1 Cache coherency protocols

Modern CPUs greatly outperform main memory regarding latency, therefore caches were placed in between the CPU and main memory to reduce access latency [HP18]. Once the CPUs accesses a block of memory, it will be stored in the cache such that subsequent accesses will be faster. These blocks of memory stored in the cache are called *cache lines*.
When multiple cores are performing actions on the same memory block, each core has their own fast cache and uses them whenever they can. However, as these cores are sharing data, they must ensure that no one is working with stale data. If some core ever modifies the data and holds it in their cache, no other core is allowed to read the data from main memory as it is outdated.
To maintain coherency between the cores, chip manufacturers invented cache coherency protocols. These coherency protocols are usually implemented using a directory or snooping based approach. In the directory-based approach, each core has access to a shared directory and updates the sharing status of a block of memory upon accessing it. Snooping-based protocols stipulate that every cache, which has a copy of a block of memory, tracks its sharing status by snooping on the broadcast medium to which each cache is typically connected [HP18].
The most common snooping protocol is *MESI*. In the MESI protocol, every actor holds a cache line in one of the states *Modified, Exclusive, Shared* or *Invalid*. Depending on the state of the cache line, each actor knows what to do upon certain events and how to *update* its status. For example, if an actor holds a cache line in a *modified* state and receives a snoop request which desires to cache the line, they must share the modified cache line with the requester *and* update the cache line's status to *shared*. Just like that, the MESI protocol defines all possible state transitions.
The major chip manufacturers Intel and AMD are using variations of this protocol, which are *MESIF* and *MOESI*, respectively [HP18].

2.2 CXL

The Compute Express Link (CXL) is an open standard defining a family of three interconnect protocols which aims to tackle limitations imposed by Peripheral Component Interconnect Express (PCIe) and Double Data Rate (DDR) interfaces [DSBB24]. It reuses the PCIe 5.0 physical layer, which makes the interconnect scales with its bandwidth. Two major limitations which motivated the development of CXL are represented in its sub-protocols.

- CXL.cache provides *coherent access to system and device memory*, this solves the problem that PCIe devices and CPUs are not able to cache each others memory.
- CXL.mem tackles *memory scalability* and *memory pooling* problems by enabling the use of memory connected via PCIe as system memory. This greatly increases the memory bandwidth per CPU and the overall memory capacity.

Devices employing CXL can decide to implement CXL.cache, CXL.mem, or both protocols, resulting in CXL type 1, 3, or 2 devices. However, all CXL devices need to implement the third protocol CXL.io, which is based on PCIe. It is responsible for many things including device discovery, status reporting, virtual-to-physical address translation, and traditional Direct Memory Access (DMA).

2.2.1 CXL.cache

CXL.cache is the sub-protocol in CXL responsible for cache coherency. Compared to traditional coherency protocols, which operate symmetrically, CXL.cache adopts an asymmetric approach. In symmetric protocols, each node implements the same coherency algorithm to track cache line states. Examples include Intel's Ultra Path Interconnect (UPI) [Int22], AMD's Infinity Fabric [AMD20], and Nvidia's NVLink [Nvi14].
In the asymmetric model of CXL.cache, the host CPU and the device maintain different views of data. The host continues to use its internal coherency protocols, e.g. *MESIF* or *MOESI*, while the device side uses a much simpler *MESI*-based protocol. This decouples CXL.cache from the host specific protocols, enabling simpler device-side implementation and better compatibility.
The interface between host and device has independent channels in both directions, device to host (D2H) and host to device (H2D). Each direction consists of three channels, named after the transactions they carry: *Request*, *Response* and *Data*
D2H requests usually target memory for reading and writing, however the protocol also allows for flushing an address (`CLFlush`), requesting exclusive ownership of a cache line

(RdOwnNoData), or indicating the eviction of a cache line from the device [CXL19, Table 15]. *H2D responses* typically contain information on the ordering of messages, request for data, or indicate the MESI state which the device should store the cache line in [CXL19, Table 11]. On the other hand, *H2D requests* are snoops to maintain coherency and indicate where data should be returned if needed [CXL19, Table 10]. *D2H responses* consequently indicate which state the requested line has in the devices cache and may announce that data will be returned [CXL19, Table 8]. Both *data* channels transfer cache lines which were either explicitly requested through read requests or implicitly requested by a snoop.
Even though the host CPU might use a more complex cache coherency protocol internally, the CXL.cache protocol relies on MESI states and is therefore decoupled to the host-specific details.
It is noteworthy that all D2H requests must use host physical addresses (HPAs). In case the device only has access to virtual Input-Output (I/O) addresses, it can issue a request via Address Translation Services (ATS) [PS19, Chapter 10] to initiate the address translation on its own. ATS is a PCIe feature which was slightly extended for CXL and is available via requests on CXL.io.

2.2.2 CXL.cache opcodes

The different type of D2H requests are differentiated using an *opcode* field in the request. These opcodes can be roughly categorized in read, write, eviction and miscellaneous requests, see Table 2.1.

Table 2.1: Categorized CXL.cache opcodes

Semantic	CXL.cache opcodes
Read	`RdCurr, RdOwn, RdShared, RdAny`
Write	`ItoMWr, MemWr, WOWrInv, WOWrInvF, WrInv`
Eviction	`CleanEvict, DirtyEvict, CleanEvictNoData`
Miscellaneous	`RdOwnNoData, CLFlush, CacheFlushed`

The read requests differ from each other in how the device wishes to cache the data locally. Write requests differ in their memory ordering guarantees, size and intend to cache data in the host cache. Moreover, we have eviction requests for both modified and unmodified cache lines.
For the miscellaneous requests we have `RdOwnNoData` which only requests ownership of a cache line without requesting data from the host. Last but not least, `CLFLush` requests to invalidate a certain cache line and `CacheFlushed` notifies the host that the device no

Table 2.2: Written out names and short descriptions for CXL.cache requests

CXL.cache opcodes	Description
`RdCurr`	Read current content with no caching intent
`RdOwn`	Read and take ownership
`RdShared`	Read and cache as shared
`RdAny`	Read and cache in any state
`ItoMWr`	Invalid to modified write
`MemWr`	Memory write
`WOWrInv`	Weakly-ordered write and invalidate
`WOWrInvF`	Weakly-ordered write and invalide full line
`WrInv`	Write and invalidate (strongly-ordered)
`CleanEvict`	Eviction of clean line
`DirtyEvict`	Eviction of dirty line
`CleanEvictNoData`	Eviction of clean lines but the device will not send the data
`RdOwnNoData`	Take ownership of line without reading content
`CLFlush`	Cache line flush
`CacheFlushed`	Notify that our caches are flushed

longer holds any cache lines in their caches.
We provide written out names for each request in Table 2.2. The written out names are not included in the specification and were deduced by us. Note that the request `ItoMWr` request is misleadingly named. Other requests like `RdShared` or `CleanEvict` are written from the perspective of the device. However, the `ItoMWr` request will not cache the line in *modified* state in the device. It will rather atomically write the cache line to the host LLC and thus the line will remain in a modified state in the host and invalid in the device.

2.3 Cache architecture

Caches are usually divided in *multiple levels* with increasing latency and size. It is common to have three levels of caches and we respectively call them L1, L2, and L3 caches. The L3 cache is the biggest and slowest cache among them and it is often referred to as LLC [HP18]. In a multi-processor system, the L1 and L2 caches are usually core private and the LLC is shared across all cores.
Modern caches are usually *set-associative*, meaning that the cache is organized in multiple *sets*. Once a block of data should be cached, a subset of the address bits is used to determine which set the data should be stored in. To be precise, if we have 2^s sets, then the lower s bits after the cache line offset are used to determine the set, we thus call them the *set index*. Then the cache line can be stored in any of the *ways* the set consists of. The cache

line offset is usually $b = 6$ because cache lines are commonly of size $2^6 = 64$ bytes. Because this was not complicated enough, the LLC cache is further subdivided into multiple *slices* which are distributed evenly across all the cores in a multi-core system. A subset of address bits are combined to compute a *slice index* which determines the target slice.

2.3.1 Inclusive vs. non-inclusive caches

We call a cache *inclusive* with respect to another cache, if the former contains a copy of every line stored in the latter cache. That means, a LLC inclusive to the L2 cache would have a copy of every line in the L2 cache. On the other hand, a non-inclusive LLC might not hold a copy for every line in the L2 cache.
There is a trend in modern server processors to design the LLC as *non-inclusive*. Even though we have not introduced cache attacks yet, it is worth noting that this change has made cache attacks on the LLC much harder [YSG+19]. But designing them as non-inclusive was publicly announced as a performance enhancement [Int24], not a cache attack mitigation.

2.4 Cache attacks

The obvious and desired effect of caches is that memory accesses take less time, when the data is present in the cache. With access to time sources like the x86 instruction `rdtsc`, we can measure the access latency and hence determine whether a line was present in the cache or not.
If we are in a cloud scenario with co-located tenants, they all share the same hardware and therefore use the same caches. Once a cache is full, a replacement policy is used to determine which data to evict next out of the cache. If a co-located tenant is excessively using the cache, we are able to detect that by measuring whether our data has been evicted from the cache. Which we can do by measuring its access latency.
There are two major classes of cache attacks, flush-based and eviction-based attacks.

2.4.1 Flush+Reload

A prominent flush-based cache attack is FLUSH+RELOAD [YF14]. The attack assumes shared memory with the victim, which would occur if, e.g., memory deduplication is enabled [RFKG25]. However, that is not the case on public cloud platforms anymore [SKLG22]. The attack forces the removal of a cache line from the cache using a *flush* instruction. As we assume coherent caches, this also affects the victim caches. We then wait and later measure how long it takes to *reload* the data. This way we determine whether

the victim accessed the data between the flush and reload operations. Low latency means that the victim accessed the data, large latency means it has not accessed it.

2.4.2 Prime+Probe

Eviction-based attacks target a cache which is shared with the victim and forces *contention* and thus cache evictions. A classic eviction-based attack is PRIME+PROBE. The attacker first *Primes* an entire cache set with its own data and then waits for the victim to access sensitive data. This set of cache lines, the attackers uses to fill the cache set, are called *eviction set*. In case the victim now allocates data in the target eviction set, at least one cache line of the attacker will be evicted. The attacker can then identify this by *Probing* the cache set to see if any access takes significantly longer, which would mean that the data was evicted and thus the victim indeed accessed the cache set.

2.5 Intel DDIO

As our experiment setup uses a recent Intel Xeon server processor, it features Intel DDIO. Classic I/O devices make use of DMA via PCIe in order to transmit their I/O data to the host. However as this DMA is not cache-coherent, the I/O data must be written to the main memory before the host can use it. While this is not a problem for many devices, network interface cards (NICs) have evolved rapidly and trips to main memory became a bottleneck. To tackle this, Intel DDIO enables devices to directly interact with the LLC, making it the main target for I/O operations. Doing so, the maximum I/O bandwidth is not limited by the memory bandwidth and furthermore the access latency to I/O data is drastically reduced [Int12]. If a device is connected via DDIO, we will call it a *DDIO-device*. Note that DDIO is a host feature and devices do not choose whether they use DDIO or not, the usage is transparent to them.

2.5.1 DDIO region

It is well known that only a subset of the LLC is accessible by DDIO-devices [KGA+20, PTV22]. This area is called *DDIO region*. Per default only 2 ways of each cache set are accessible to DDIO-devices, however the number of accessible ways can be configured in the `IIO_LLC_WAYS` model specific register (MSR) [PTV22]. Additionally, [PTV22] has found a previously unknown region which they call the DDIO$^+$ region. However, even with the DDIO$^+$ region, a DDIO-device is not able to access all ways in a cache set. Thus, a device is not able to perform contention based cache attacks like PRIME+PROBE, as priming the entire cache set is not possible.

2.5.2 Related work

Allowing devices to directly access the LLC begs the question whether an attacker can misuse this to performor assist micro-architectural attacks. Prior research has shown that heterogeneous attackers, which have control over a CPU core *and* a DDIO-device, can accelerate attacks like Rowhammer [WTM+20] and significantly ease cache attacks [PTV22]. Cache attack become easier because with a DDIO-device, an attacker can *monitor* the state of the cache without modifying it and *accelerate* LLC eviction set construction, even for non-inclusive LLCs [PTV22].

2.6 Common Intel terminology

As our experiments are conducted on an Intel FPGA, let us cover some common FPGA terminology and Intel specific terms. An FPGA is reprogrammable and the configuration file is vendor-independently called a *bitstream*. When working with Intel FPGAs, one will quickly come across the term *Accelerator Functional Unit (AFU)*. The AFU is the part of a design which is usually written by the user to implement their custom logic. This term is originally tightly coupled to Open Programmable Acceleration Engine (OPAE) platforms. However, from our experience this term is also used on non-OPAE platforms to highlight modules which the user can modify or interact with.
Moreover, one might have seen the terms *green bitstream* and *blue bitstream*. The green bitstream contains a user design whereas the blue bitstream contains logic to interact with the FPGA specific I/O peripherals and other proprietary logic. The green and blue bitstream communicate via defined interfaces.
We will not use those terms and rather distinct between the *Example Design* and *CXL IP*. More on that in Section 3.3.

3 Experimentation Setup

This chapter showcases our setup including hardware components, IP blocks, tools and protocol versions.

3.1 Hardware

To exercise the CXL.cache protocol, we need some sort of hardware capable of speaking the protocol. Our host machine has a Supermicro X13SEI-TF motherboard and is equipped with a 16 core 5th Gen Intel Xeon Processor (SILVER 4514Y) supporting CXL 1.1 [Int23]. Regarding the caching subsystem, the processor has a 16-way set-associative core-private L2 cache of 2 MiB and a 15-way set-associative shared LLC of 30 MiB. This LLC is non-inclusive, meaning that not all data present in the L2 cache has to be present in the LLC.
We use the Mercury A2700 Accelerator Card [Ter23] to realize our CXL.cache device. The heart of the accelerator card is an Agilex 7 I-Series FPGA (AGIB027R29A1E2VB) which contains a so called R-Tile transceiver that is said to support the CXL 1.1 and 2.0 specification [Alt25a].
The choice to use an FPGA for our experiments was partly driven out of necessity, as there are no commercially available devices yet besides FPGA accelerators which support CXL.cache. But there are already numerous memory expansion devices utilizing CXL.mem [Mic25a, Sam25].

3.2 Host software

Our host does not need additional software to support CXL.cache, accesses on the CXL.cache layer are fully transparent to the Operating system (OS). However, one has to allow CXL devices to perform CXL.cache requests by setting the *Device Trust Level* in the unified extensible firmware interface (UEFI) to *Trusted CXL Device*. This access control is defined in the CXL specification [CXL19].
A CXL 1.1 device will appear to the OS as a PCIe Root Complex Integrated Endpoint (RCiEP), which is different to CXL 2.0 where the device will be exposed as a PCIe endpoint. Thus any communication with the device for configuration purposes can be done via PCIe Memory-mapped I/O (MMIO) registers.

3.3 Development

We are using Quartus Prime Pro 24.3 to program the FPGA. As we only want to analyze CXL.cache, a CXL type 1 device (CXL.cache + CXL.io) is sufficient to us.
Because developing a working CXL controller is itself worth a publication [FLC+25], we make use of a pre-built hardware module shipped with the installation of Quartus Prime. The full name of the IP is *R-Tile Intel FPGA IP for Compute Express Link (CXL)*.
However, interacting with the CXL IP itself is complex as well, since we would need to study the full interface to the IP, which consists of hundreds of signals, and adapt our user logic to work with that. To prevent this development overhead, most big IPs come together with an *Example Design* including an *User Guide* which demonstrates an example usage of the IP. These Example Designs are usually designed in a way to leave space for additional user custom logic. This way, we as the users just need to interact with the minimal, well documented interface provided by the Example Design.

4 Working without Documentation

Usually working with an Example Design of an IP is straightforward. Quartus Prime has a reference to the corresponding User Guide, which then contains all the information a developer needs to understand how the Example Design works. Among other things, it explains the directory structure, interface to the IP, and potential places to extend the Example Design.
However, as we learned the hard way, its different for the CXL IP. The User Guide is restricted to people who have signed a non-disclosure agreement (NDA) [Alt25b]. Since we do not have such an agreement with Intel, there is no way for us to access the User Guide. We reached out to Intel about this at the beginning of our work, but the employees we have talked to could not share the file with us since its marked as *internal*.
Due to that, making use of the Example Design turned out to be more difficult than expected, since we have to rely on code audits to understand the codebase.

4.1 Struggles

In the following sections we present the two major problems we ran into as we are working without documentation.

4.1.1 Where to add our logic?

Our goal is to determine whether an attacker is able to perform cache attacks using CXL.cache. To do this, we need to issue D2H requests and analyze their behavior. Without having access to any documentation, we are left in the dark about *where* to add our logic and *if* the CXL IP even allows for issuing raw D2H requests.

To give one simple example how this hindered us: As we searched through the Example Design to find a module which we can add our logic to, we stumbled upon a promising `afu` directory. It contains only two SystemVerilog source files: `afu_csr_avmm_slave.sv`, `afu_top.sv`. Whereas `afu_top.sv` is a top module which would instantiate and connect other modules in a more complex design. Looking into the code we even find a comment stating `User can implement the AFU logic here`. Which makes us think that this is the module we are supposed to modify. However, when auditing the root module of our Example Design, we find that this AFU module is never initialized,

therefore the AFU module is never used. At this point it was unclear to us whether this is simply a mistake of the Example Design or whether this is intended. It later turned out that the purpose of this AFU module is to capture requests to the memory controller, which can be used to, e.g., implement memory encryption. Since we generated an Example Design solely for usage of CXL.cache, our design does not contain a memory controller because that is only needed for CXL.mem. Therefore, the AFU module is never initialized, but all the surrounding code is kept in the Example Design. Without having access to documentation we had to find this out by manual code audit.

4.1.2 What is the IP hiding from us?

After further digging around in the Example Design, we find that the IP adds layers of abstractions to the D2H requests. We explain this in greater detail in Section 5.1.3. To cut the long story short, the handling of D2H requests is fully handled inside the IP, which makes them transparent to the CXL device. The only requests a CXL device can send to the IP are *Read* and *Write* request with caching hints, telling the IP in which cache state we desire to cache the data.
But as we attempt to analyze the protocol in this thesis, the added abstraction layers pose an obstacle since we cannot be sure which request the IP will issue on the CXL.cache layer. At this point, one might expect, that we can just look into the source code of the IP to understand its inner workings. But, as the name *Intellectual Property* suggests, IP vendors are eager to keep their designs a well-preserved secret. Therefore, the IP is shipped fully encrypted which obviously prevents us from performing a code audit of it.

4.2 Solving our problems

As the priorly described problems impair our work to a not negligible amount, we first focus on tackling them.

4.2.1 Large language model assisted code audit

We accelerate the initial understanding of the generated Example Design by utilizing a large language model (LLM). Specifically, we use the agentic coding tool Claude Code by Anthropic in their Max plan [Ant25]. Our workflow consists of providing selected files or directories from the Example Design to Claude Code and asking it questions about the codebase. It is well known that LLMs are prone to mistakes and hallucinations which we experienced first-hand. But this technique still greatly assisted our manual audit and was especially helpful in mentally dissecting a hardware module into its submodules or

obtaining a high-level description of a module's functionality. Moreover, the LLM made tracing data flow between multiple modules much easier without relying on static analysis tools.

4.2.2 Circumventing the IP encryption

One might wonder how the synthesis of our hardware design even works if the IP cores are stored in a fully encrypted format. To perform the actual synthesis, the EDA tool first decrypts the IP files in memory before it can process them. But since we have control over the execution environment, we can choose to monitor the execution of the EDA tool and thus get access to the secret key used to decrypt the IP files. Therefore, the protection of the IP solely relies on the obfuscation mechanism employed by the EDA tools to make reverse engineering of it much harder. But prior work has shown that the majority of EDA tools do not make use of heavy obfuscation [SSE+22] which renders reverse engineering and thus extraction of the keys feasible. The authors of [SSE+22] were also able to extract the secret decryption keys out of Quartus Prime and went into a responsible disclosure process with Intel to share their findings. However, as we reverse engineer Quartus Prime almost four years after their publication, we can safely say that the situation has not changed for the better and reverse engineering the software was very feasible. We will not share any details on the process of extracting the decryption keys from the EDA tool due to an ongoing disclosure process with Intel.
Having extracted the decryption keys, we proceeded to decrypt the IP files and are able to audit the source code whenever we are in need of understanding its inner working.

5 Platform Investigation

In this chapter we showcase our modifications of the Example Design in order to perform our experiments and how the interface to the CXL IP core is designed. We focus on how CXL.cache requests can be issued and to which extend the IP limits us to directly interact with the CXL.cache protocol.

5.1 Interface to the IP

Understanding the interface was of major interest as we need to know which bus to use for our requests and most importantly what data the IP expects on that bus.

5.1.1 How we modified the Example Design

We are searching for how to modify the Example Design in order to issue our own requests to the CXL IP. Looking for the string *AFU* in the entire Example Design results in many occurrences and going through them yields one promising hit. We find a module instantiated at the top of the Example Design which is commented with `// User AFU`. The module instantiated after the comment is guarded by a compile-time macro and depending on its value, we can either synthesize an example AFU (`afu_atomic_test_engine`) or a custom user implementation (`cust_afu_wrapper`). As it turns out, the module `cust_afu_wrapper` likely stands for *Custom AFU wrapper* and is merely an empty template for us to fill. Because we are working without any documentation, we first study the example AFU (`afu_atomic_test_engine`), which we will abbreviate as ATE.
The Atomic Test Engine (ATE) has a simple architecture, it consists of some In-/Output connections from registers addressable through MMIO from the host and one *Advanced eXtensible Interface (AXI)* interface to the CXL IP. To be precise, it is an *AXI4-MM* interface, which is a bus protocol commonly used to connect multiple IP blocks. For our use case, all we need to know is that AXI allows us to issue Read/Write requests to a memory address together with some custom signals, which are implementation specific and not defined in the AXI specification. Please note that the memory addresses are referring to host physical addresses.
In the Example Design, the ATE module is meant to showcase how atomic write operations can be performed. Requests are issued by the host configuring the module through PCIe MMIO. Since the module already includes a state machine, configuration registers,

and other logic to successfully issue requests to the IP, we will continue to modify it, instead of writing a module from scratch. The modifications to run our experiments can be summarized as follows:

- Implement a timer to measure the cycles taken for requests to finish.
- Extend the finite state machine with states and logic to issue read requests and arbitrary write requests.
- Adapt existing configuration registers to dynamically change request types, e.g., Read/Write.

5.1.2 Request format

We mentioned before that the AXI bus used by the CXL IP contains custom user signals. In order to understand how these requests look like, we will present the custom signals. The following SystemVerilog struct is used to specify a request's type [1]:

```
1 typedef struct packed {
2   logic                       target_hdm;
3   logic                       do_not_send_d2hreq;
4   t_cafu_axi4_awuser_opcode   opcode;
5 } t_cafu_axi4_awuser;
```

We specify whether our request targets host-managed device memory (HDM) and can choose not to issue a D2H request. Note that not sending a D2H request only makes sense if we issue a request to device local memory. If we send a request to host-attached memory via CXL.cache, a D2H request will be issued nonetheless. For our use case, we will always target host-attached memory and issue D2H requests. What matters the most to us are the opcodes, which we will refer to as *AXI opcodes*.

Looking at the write opcodes defined in the Example Design, it is clear that there is no obvious mapping from these opcodes to the write opcodes specified in the CXL.cache specification. This is highlighted in a direct comparison in Table 5.1. A simplified overview of the CXL.cache opcodes is given in the Background 2.2.2.

[1] Please note that this is the struct used for the AXI *Write* channel, however the format for the *Read* channel is identical, it just uses different opcodes. Path in the Example Design: `common/cafu_csr0/cafu_common_pkg.sv`

Table 5.1: Overview of Write opcodes as defined in the CXL specification [CXL19] versus the Write opcodes existing in the Example Design.

CXL.cache opcodes	AXI opcodes
ItoMWr	eWR_I_WO
MemWr	eWR_M
CleanEvict	eWR_I_SO
DirtyEvict	eWR_BARRIER
CleanEvictNoData	eWR_EVICT
WOWrInv	eWR_FLUSHHOSTCACHE
WOWrInvF	eWR_FLUSHDEVCACHE
WrInv	

5.1.3 Request flow

Trying to determine the semantics of the different AXI opcodes, we trace the flow of the signals into the CXL IP and stumble upon the following comment in the top module: `//AXI <-> AXI2CCIP_SHIM <-> CCIP`.

What this tells us is that the request opcodes are likely being mapped to a protocol named Core Cache Interface Protocol (CCI-P) inside the IP. This protocol is used in Intel FPGA designs to access host data cache-coherently [Int19]. It merely acts as an abstraction layer to reduce the complexity of developing a user AFU and enables interoperability across multiple interfaces like PCIe or UPI. With CCI-P, each read and write request is performed with a *caching hint*, signaling in which state we want to receive a cache line. For example, the CCI-P request `RdLine_I` will read an entire cache line and result in the data remaining *invalid* in our FPGA cache, thus the line will not be cached.

Comparing the list of AXI opcodes and available CCI-P opcodes reveals some kind of connection, this is shown in Table 5.2. Note that the tables entries with a *non-existent* field still have a corresponding translated opcode. They are just not defined in the CCI-P Reference Manual [Int19]. We refer to this version of CCI-P, extended with requests for ownership gaining reads, cache line evictions, and flush operations, as *CCI-P*$^+$.

After translation to the intermediate CCI-P$^+$ format, the requests will finally be converted to corresponding CXL.cache opcodes. A simplified translation table, assuming full cache line accesses, is given in Table 5.2. Please note that we do not provide the new CCI-P$^+$ opcodes since they are only listed in the encrypted IP files which we got access to via reverse engineering.

Table 5.2: Mapping of AXI opcodes to CCI-P requests and simplified mapping to CXL.cache opcodes assuming full cache line accesses

AXI opcode	CCI-P opcode	CXL.cache opcode
eWR_I_WO	eREQ_WRLINE_I	WOWrInv
eWR_M	eREQ_WRLINE_M	RdOwnNoData
eWR_I_SO	eREQ_WRPUSH_I[2]	ItoMWr
eRD_I	eREQ_RDLINE_I	RdCurr
eRD_S	eREQ_RDLINE_S	RdShared
eRD_EM	*non-existent*	RdOwn
eWR_EVICT	*non-existent*	Varying Evict Opcode
eWR_BARRIER	*non-existent*	—
eWR_FLUSHHOSTCACHE	*non-existent*	Multiple Evictions
eWR_FLUSHDEVACHE	*non-existent*	Multiple Evictions

The last three opcodes in Table 5.2 stand out as they cannot be directly mapped to CXL.cache requests. The barrier is used as a memory fence and will not issue any requests as it only waits for pending memory writes to finish. Moreover, the FLUSHHOSTCACHE and FLUSHDEVCACHE requests walk through the FPGA's local cache and issue individual evict requests for every line belonging to *host* or *device* memory accordingly.
Note that the table is simplified and the requests being sent depend on the current state of the cache line. For example eWR_EVICT will issue either a *Clean* or *Dirty* eviction if required. Moreover, if a cache line already present in *Exclusive* state, there is no need to issue a RdOwn request when performing eRD_EM. The simplified flow of requests is sketched in Fig. 5.1.

5.2 Limitations

Using the current state of the CXL IP to develop an attack limits the attack capabilities as highlighted in the following sections.

5.2.1 Not implemented CLFlush request

The CXL specification defines a CLFlush operation which invalidates the specified cache line. It turns out that this operation is not yet implemented in the CXL IP. That way we knew that we do not have to further investigate this request. Without such insights to the

[2]We know that eWR_I_SO is mapped to eREQ_WRPUSH_I due to comments in the Example Design. We furthermore confirmed this by auditing the decrypted IP files and through our experiments.

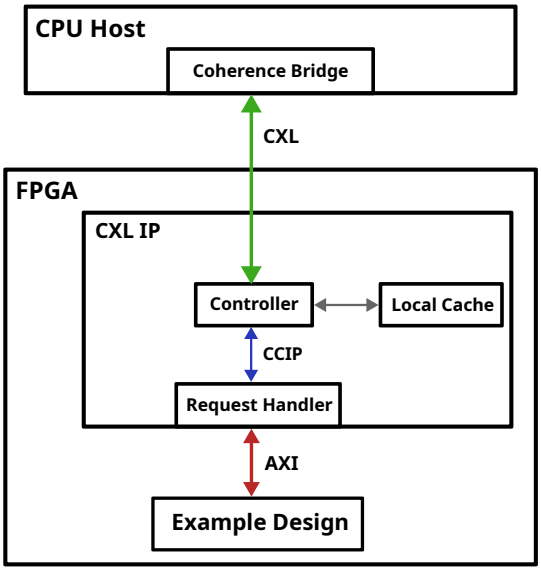

Figure 5.1: Simplified flow of requests issued from the Example Design.

IP, we would have wondered why `CLFlush` requests from the Example Design are not having any effect.
However, we strongly assume that an attacker's capabilities are not limited by a missing `CLFlush` instruction. Assuming a FLUSH+RELOAD attack, an attacker can perform the *flush* by issuing a CXL.cache `RdOwnNoData` request as this request also invalidates the specified cache line in the host cache. The *reload* phase of the attack can be performed by issuing repeated CXL.cache `RdCurr` requests. We will later see that an attacker can determine whether the line was cached in the host by observing the request latency.

5.2.2 Abstraction layer added by CXL IP

Tighter interaction with the caching protocol yields benefits for an attacker as we discuss in Chapter 6. But if a cache attack is developed using the unmodified CXL IP as a foundation, we face some limitations induced by abstractions. One of them is that fine control over issued CXL.cache requests is missing since we need to rely on the AXI opcodes as an intermediate layer. However, what limits our monitoring capabilities even greater is that handling H2D snoop requests is done inside the IP. If we had access to the snoops, we could distinguish between host read and write requests. We discuss this in Section 6.2.

6 Findings

Having everything set up and being able measure latencies of both read and write requests, we perform our experiments. Our findings and experiments are outlined in the following sections.

6.1 Bypassing the IOMMU

The most important finding for deploying CXL.cache in the public cloud is that all CXL.cache requests bypass the IOMMU by nature. All requests use HPAs to specify which memory to access instead of I/O virtual addresses which would allow for memory isolation using the IOMMU. During our experiments we could confirm that arbitrary data can be accessed even though the IOMMU is strictly enabled with the following Linux kernel parameters `iommu=on iommu.strict=1`. Note that our device is still assigned to an IOMMU group which we confirmed via the `sysfs` pseudo filesystem, because the device may need to perform virtual to physical address translations via ATS on CXL.io if they are assigned a virtual address space.
As there is no fine-granular way to restrict the access of a CXL.cache device to the address space, we conclude that CXL.cache can not be used securely by tenants in a multi-tenancy cloud scenario. The only restriction we could configure on our platform is the *Device Trust Level* documented in the specification [CXL19, 7.2.2.1.14]. However, this configuration applies to all CXL devices connected to the host and does not allow for fine-granular access control. There are three options documented which either grant *full* access to all memory, restrict access to device attached memory, or deny all CXL.cache requests.

6.1.1 Impact on threat model

This finding makes our assumed threat model of tenants using cache attacks for data exfiltration less realistic. However, as there is an interest for hardware manufacturers to get their CXL.cache accelerators into the public cloud, it is reasonable to assume that there is ongoing research on how to achieve memory isolation. Due to that, investigating the feasibility to use CXL.cache for cache attacks remains valuable. We discuss potential isolation mechanisms in Section 7.1.3.

6.2 Own+Wait

Being tightly integrated in the cache coherency protocol yields some benefits and new attack scenarios which we summarize under OWN+WAIT. Assume that we have an attacker with full access over the CXL controller. This attacker first requests ownership of cache line using `RdOwnNoData` and then simply waits for incoming snoop requests from the host. Whenever the victim accesses the target cache line, the host will send a snoop request to maintain coherency. As the attacker handles these snoops, they know that the victim accessed the line without having to continuously probe the cache line.
Note that we have not tested this technique as it requires modification of the CXL IP which was left out due to time constraints. Such a modification also requires re-encrypting the IP files, which can be done without additional reverse engineering of Quartus Prime, since a symmetrical (AES) is used for encrypting the IP files.

6.2.1 Distinguish reads and writes

One advantage of this approach is that attackers can distinguish between read and write accesses which is not possible with traditional cache attacks. The type of snoop request a host issues gives away the type of access. There are only three type of snoop requests, `SnpData` which is usually sent upon read requests, `SnpInv` which aims to invalidate the cache line in the device for write requests from the host, and `SnpCurr` which is only sent if another CXL.cache device issues a `RdCurr` D2H request [CXL19, 3.2.4.3.3]. With that we can distinguish between read and write requests.

6.2.2 Partially evading cache attack detection

As cache attacks became popular, with it came research for countermeasures. Because many proposed countermeasures, both hardware and software, would impose a significant computational overhead, the idea of only low overhead detection mechanisms gained attention [KFC+24]. These detection mechanisms often leverage Hardware performance counters (HPCs) to collect information on micro-architectural events like cache misses [KFC+24], even though their effectiveness for reliable attack detection is controversial [KFC+24]. Detection mechanism can be categorized in detecting *attacker*-events [CFQP23, MAPMK+20] or *victim*-events [BPSS24, BIME18, WMYZ22]. Victim focused detection mechanisms observe whether the victim, e.g., frequently encounters cache misses when accessing secret data. On the other hand, interesting attacker events could be clean evictions that occur abnormally often in PRIME+PROBE attacks [MAPMK+20].
All detection mechanisms which assume the attacker to be on the CPU and thus attempt to detect attacker-events fail to detect an attacker using CXL.cache as the attack is performed

by an I/O device. Victim-based detection mechanisms still work as expected which we verified by using HPCs to collect cache misses of a victim as a CXL.cache device repeatedly invalidates the memory line in the victim caches using `RdOwnNoData` requests.

6.3 Determine the state of a host cache line

An attacker can also determine the state of a cache line in the host caches by actively issuing requests and observing the resulting response. This is a prerequisite to perform cache attacks [RFKG25].

6.3.1 Received MESI state

D2H requests which aim to cache a line need to be told in which MESI state they receive the cache line. The device is notified about this state in the H2D response to their request. If the device now reads a cache line using the `RdAny` opcode, they can receive the cache line in any MESI state, as highlighted by [CXL19, Table 16]. For example, receiving a line in *exclusive* state in response to a `RdAny` request indicates that the host should currently not hold a copy of that line in their cache. Likewise, if the host currently holds a copy of a cache line, the device will likely receive the line as *shared*. Note that the specification allows for the device to receive a cache line in *modified* state [CXL19, Table 16], which might be the case if the host currently holds a modified copy of the cache line which has not been written back to main memory yet. However, we have not tested our assumptions regarding the `RdAny` opcode as this request can not be issued 5.2 without IP modification and observing the MESI state in the H2D response also requires IP modifications. These modifications were left out due to time constraints.

6.3.2 Timing latencies

Beyond determining if a cache line is currently present in the host cache, an attacker can locate where the line is present in the cache hierarchy by observing the request latency. This monitoring is possible without modifications to the IP as only a hardware counter needs to be added to the Example Design in order to measure the latency.

To show that the request latency varies with the state of the cache line, we prepare a cache line in a certain state and then issue a `RdCurr` CXL.cache request targeting that line. We measure the latency on the CXL device by counting the clock cycles until the request is completed.

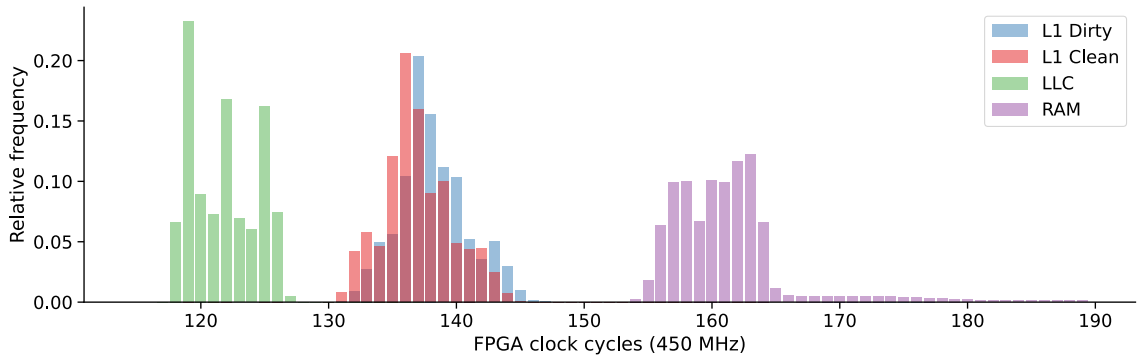

Figure 6.1: Request latency of CXL.cache `RdCurr` requests with data being present in certain states

Our experiment performs the following steps:

- Allocate a new page of memory and select a cache line.
- Prepare the state of the cache line by, e.g., accessing or flushing it.
- Issue a `RdCurr` request to the cache line and measure its latency. The `RdCurr` does not cache the data in the devices cache nor does it modify the state of the hosts cache [CXL19, 3.2.4.1.6].

We perform the experiment with four different cache states.

- *Flushed*: The line is evicted from the host caches using the `clflush` instruction.
- *L1 cache clean*: The line is placed in the host L1 cache by reading from it.
- *L1 cache dirty*: The line is placed in the host L1 cache by writing to it.
- *LLC*: The line is placed in the hosts LLC by using the `ItoMWr` CXL.cache opcode.

Note that these preparations are mostly performed on the host itself. Only the `ItoMWr` request is performed by the CXL.cache device. Each experiment is repeated two million times and memory fences (`mfence`) were placed accordingly to ensures accesses to memory are completed before issuing CXL requests. We furthermore alternate between the tested cache state on each access to reduce the impact of noise.
As seen in Fig. 6.1, different cache levels can clearly be distinguished using the request latency. All results seem reasonable besides that accesses to the LLC end up being faster than accesses to the L1 cache. We will explain that in Section 6.4. Moreover, it is also reasonable that accesses to clean or dirty lines in the L1 cache do not show major differences.

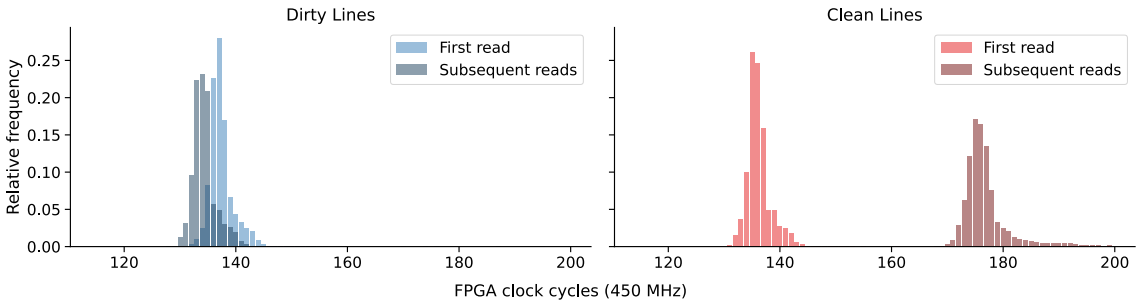

Figure 6.2: Differences between the latency of the first and subsequent `RdCurr` requests if data resides clean or dirty in the L1 cache.

However, we noticed a major difference regarding clean and dirty lines if the CXL device issues *multiple* subsequent requests to the *same* cache line. Starting from the second request, we are experiencing unusual large latencies. This is highlighted in Fig. 6.2 which compares the latencies of the first request against subsequent accesses for both clean and dirty lines in the L1 cache. Subsequent accesses to clean cache lines have access latencies larger than requesting data from main memory. This behavior is not observable for dirty cache lines and can thus be used to distinguish clean and dirty cache lines.
We cannot explain this behavior. Potential explanations for this were evictions from the L1 cache triggered by the `RdCurr` request or risen errors due to subsequent reads. However, the specification does not describe this behavior and we could verify that the data remains cached in the host L1 cache after `RdCurr` requests by measuring the host access latency using `rdtsc`. Moreover, the specification states that subsequent reads to the same cache line are allowed [CXL19, 3.2.5.6]. Unlike the first clean read, subsequent clean reads show a pattern similar main memory responses. A fraction of requests are delayed by ten cycles or more which can be seen in Fig. 6.2 with the bars slowly fading out to the right.
It is noteworthy that the gap between the two spikes have a consistent distance of approximately 40 cycles throughout our testing. Interestingly enough, accesses to the local cache instantiated by the CXL IP take exactly 46 cycles which makes us think whether there is an implementation flaw in the CXL IP which causes unnecessary accesses to the local cache before issuing the request to the host. We discuss in Section 7.1.2 how this can be further investigated.

6.4 Direct access to LLC

Accesses to the LLC from CXL devices take less time than accesses to data located in the L1 cache, as shown in Fig. 6.1. This aligns with related work which demonstrates how cache attacks can be assisted by PCIe devices on system with Intel DDIO [PTV22]. While it is

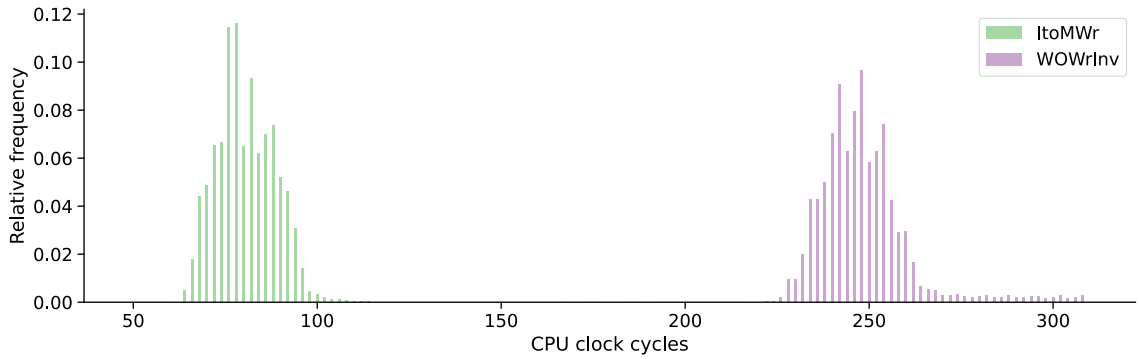

Figure 6.3: Access latencies for CPU reads to previously uncached lines written via CXL.cache `ItoMWr` or `WOWrInv`. Demonstrating that `ItoMWr` allocates the line in the host LLC while `WOWrInv` does not.

known that DDIO enables PCIe reads to be directly served from the LLC, it is unknown whether Intel Xeon processors use DDIO to serve CXL.cache requests or handle them differently.

6.4.1 Reading cache lines without altering state

With DDIO in use, read requests will be served through the LLC, however if a line is not present in any cache, it will be served from main memory without allocating the data in the hosts caches [PTV22]. We could verify that cached data is directly served from the LLC for CXL.cache reads as shown in Fig. 6.1. Furthermore, we verified that CXL.cache read requests do allocate previously uncached data in the host caches. We did that by reading an uncached line from CXL.cache and measuring the hosts access latency to that line.

6.4.2 ItoMWr targets LLC not main memory

The key benefit of DDIO is that DMA writes can be directly written into the hosts LLC. We found that there is a CXL.cache write request which targets the hosts LLC instead of main memory, that is `ItoMWr`. The CXL specification does not explicitly state that, but comparing the opcodes description with the related request `MemWr` it becomes clear that `ItoMWr` will always write the cache line into the hosts cache. We verified this assumption by measuring the hosts access latency to a cache line after writing to it using either `ItoMWr` or `WOWrInv`. The difference in Fig. 6.3 is clear and the cycle counts align with separately acquired measurements for the LLC and main memory.

6.4.3 Access to entire cache set

We found that `ItoMWr` requests can target all ways in an LLC cache set. This is a finding incompatible with known behavior of DDIO since DDIO limits the amount of ways a device can write to [PTV22, WTM+20]. Just like [PTV22], we denote the amount of accessible cache ways as D. Per default a device can only access two cache ways per cache set ($D = 2$) which can be verified and changed using the MSR called `IIO_LLC_WAYS`. This way we verified that D is two on our system.

To confirm that we can write to the entire cache set independently of the setting of D, we perform the following experiment:

- Write to the target cache line using `ItoMWr` which allocates it in the host LLC.
- Write to many cache lines using `ItoMWr` until we encounter an eviction of the target cache line.

Note that this is the problem of finding *eviction sets* [VKM19]. Eviction set finding is quite easy with access to CXL.cache, even on systems with non-inclusive caches because we can directly write to the LLC and check for the existence of a line in the cache without modifying its state. Moreover, we have access to physical addresses and can therefore construct addresses with matching LLC set indices. The only thing we cannot reliably determine is which LLC slice an address is mapped to.

However, assuming the addresses are evenly distributed across the slices, 16 slices with 15 ways per cache set give an expected eviction after accessing about $16 \cdot 15 = 240$ addresses [VKM19]. Putting this assumption to test by writing to our candidates using `ItoMWr`, we see in Fig. 6.4 that we indeed need about 230 writes until the target is evicted, thus filling the entire cache set. If the restriction to only two cache ways ($D = 2$) was effective, then we would have seen an eviction of the target cache line after about $16 \cdot 2 = 32$ writes. Note that we did not separately confirm whether DDIO correctly limits regular DMA accesses to only D ways. So there remains a small chance that this is a false finding that CXL accesses have greater power than traditional DDIO accesses.

Due to the similarities to DDIO and gained insights from inspecting the IP files, we conclude that the direct cache access via CXL.cache is probably realized with an adopted version of DDIO which allows for unrestricted access to all cache ways. By having access to all ways of the cache set, we can perform PRIME+PROBE attacks which is usually not possible for devices using DDIO [WTM+20].

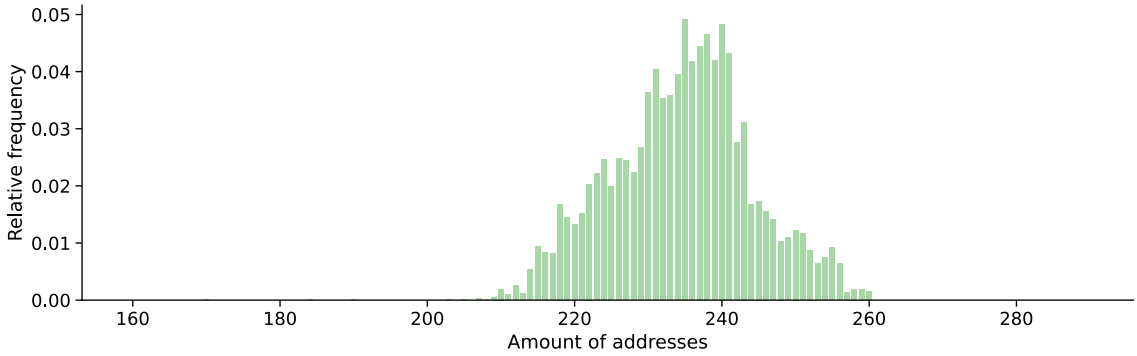

Figure 6.4: Amount of addresses written to using `ItoMWr` until the target is evicted.

6.5 Impact of Device Trust Level

As mentioned in Section 6.1, there is a *Device Trust Level* register which allows for the following configurations:

- Fully Trusted: Access on both host-attached and device-attached memory.
- Semi-Trusted: Restrict access to device attached memory ranges only.
- Untrusted: All accesses on CXL.cache will be aborted by the host.

We could verify that these settings block the device from accessing host memory as expected. Moreover, we tested whether the *semi-trusted* configuration allows *cross-device* access to CXL.mem memory. While the specification clearly documents that the semi-trusted setting restricts access to device attached memory, it is not clear whether a device is only allowed to access its own memory or also from other devices which provide memory via CXL.mem. Because we have a separate thesis working on CXL.mem, we have a CXL.mem capable device and could verify that cross-device accesses are *not* possible using the semi-trusted configuration.
However, this configuration is coarse grained and will be applied to *all* CXL.cache requests, independently of which device issued the request. This way we need to block access to CXL.cache for all device once we have a single untrusted device in the system.
While the specification states that the host can still protect security sensitive memory regions even if we access to host memory is allowed, it does not state how that should be done. On our system, the UEFI contained an additional description saying that pages marked as *uncachable* can not be accessed. We only investigated this on the side by trying to access the physical address corresponding to the config space of a PCIe device. Trying to access the address directly using CXL.cache lead to the machine shutting down without any error messages in the kernel log. We leave it to future work to further investigate this.

6.5.1 Supply-chain attack potential

Because the CXL subprotocols CXL.cache and CXL.mem share a transaction and link layer, every device capable of sourcing CXL.mem requests should be able to issue CXL.cache requests as well. We need to be aware of that if we are using *untrusted* CXL.mem expansion boards. Once we allow accesses on CXL.cache, these devices could also try to access host memory. This might be used for data exfiltration in a cloud scenario where the victim has access to the memory expansion board and could trigger data exfiltration via specific access patterns to the expansion board.
Note that this just outlines the feasibility for such a supply-chain attack. While we do not expect this to become a threat to regular cloud users, this is something to consider for cloud services processing sensitive data.

6.6 Intel TDX

When deploying a classic virtual machine (VM) in a public cloud we trust the operator of the cloud center to not access or mess with our data. However, Intel Trust Domain Extensions (TDX) [Int25b] allows to deploy VMs which provide data integrity and confidentiality even in the presence of an evil hypervisor. Thus the hypervisor and, e.g., malicious I/O devices are included in the threat model. These protected VMs are called Trusted Domains (TDs) and are managed by the so called *TDX module* which is an Intel-signed software module.
As CXL.cache offers a new and unique way to access host physical memory, it begs the question whether it can be used for new attacks against TDX.

6.6.1 Naive memory reading

Memory of a TD is split into *shared* and *private* sections. The private sections remain encrypted in main memory and are supposed to be only readable by the TD itself. The CPU uses the newly introduced Secure Arbitration Mode (SEAM) to track whether an access to private memory is allowed. Read requests issued outside of the SEAM mode should return only zeros [COV+24].
Since CXL.cache devices have somewhat direct access to the host LLC (Section 6.4), we want to test whether CXL.cache read requests will also read all zeros if they target TD private memory. To perform this experiment, we need a HPA of TD private memory. This can be done using the interface to the TDX module to get the translation for a guest physical address (GPA). However, due to our inexperience with the TDX ecosystem and time constraints, we did not follow this path.

Instead, we leverage that the physical memory regions which can be used for allocating private TD memory is printed in the kernel log. These regions are called *convertible memory regions* and in our case span across nearly all of our available main memory. If we iterate over all these pages, we will consequently have also iterated over the private memory regions. Even if only a small quarter of them are actually used as private TD memory.
We perform the following experiment for all pages in the convertible memory regions after starting a TD. Our system uses a Linux kernel version 6.8.0 and a TDX module of version 1.5.

- Pick the first cache line of the page.
- Read from it via a kernel module[3] and CXL.cache using `RdCurr`.
- Compare if the results are different.

The result of this experiment was that the kernel module and CXL.cache device read the *same* values for all accesses. Thus, accesses via CXL.cache do not trivially get access to the ciphertext of a TD. While this experiment is quite simple, we do not see a reason for it to be unsound. We discuss steps for further research in Section 7.1.4.

[3] `https://github.com/NateBrune/fmem`

7 Conclusions

In this thesis we investigated the security implications when using CXL.cache suitable hardware in the cloud and outlined interesting research directions for future work.

7.1 Discussion and open problems

In the following we discuss some of our findings and outline ideas for future work.

7.1.1 Own+Wait

While it is feasible to distinguish between read and write requests by observing incoming snoop requests, this type of monitoring requires direct access to the target cache line. If an attacker already has access to the physical page, they can just read the cache line and therefore observe whether the victim modifies the cache line. Therefore, comparing this technique to regular cache attacks is slightly unfair since our attacker already has greater capabilities. However, the general technique of monitoring incoming snoop requests still remains useful for shared memory scenarios.

7.1.2 Additional delay for clean lines

We found that subsequent CXL.cache reads to a clean cache line add a significant overhead of around 30 percent, as seen in Fig. 6.2. It stands out that the additional delay is almost as large as an access to the local device cache which takes 46 cycles.
To further investigate this, one could modify the IP files to monitor accesses to the local cache and install a timer in the module which ultimately sends out the requests. This way one could determine whether the additional delay is coming from the host or whether the time is spent in the IP itself.

7.1.3 Memory isolation

It is likely that more devices will adopt CXL to coherently access host memory in the future. But without memory isolation, CXL.cache requests have to be blocked in a cloud scenario and thus the capabilities of the hardware are not fully utilized.

Isolation mechanisms could be implemented on the device side by, e.g., forcing the usage of I/O virtual addresses which are translated using CXL.io. However, if a device manufacturer messes up the isolation this can directly lead to a system compromise. Therefore, such isolation should rather be implemented on the host.
The host processor could use a mechanism similar to RISC-V's physical memory protection (PMP) [RIS25]. This mechanism could be adapted to allow configurable privileges for each CXL port connected to the CPU. We leave further thoughts on this for future work.

7.1.4 Intel TDX

Instead of naively trying every physical address, one should rather use the TDX module interface to translate a target GPA to a HPA and repeat the experiment. This translation can be done using the `TDH.MEM.SEPT.RD` `seamcall` [Int25a].
Additionally, future work should investigate how memory writes into a TD's private memory are handled. The specification states that each cache line is marked with a *TD Owner bit* which is stored in the error correction code (ECC) memory. Processes running outside the TD will clear the TD Owner bit upon writing to such a cache line, thus alerting the TD that its data was messed with. If a CXL device uses `ItoMWr`, the write request targets the LLC and the consequences for the ECC bits should be examined.

7.2 Summary

To explore how CXL.cache could be used for cache attacks, we examined an FPGA board with the goal of using Intel's CXL IP for our experiments. First, we utilized an LLM to assist our code audit of the CXL.cache template project and circumvented the IP encryption to account for the lack of accessible documentation. While this not only assisted us to start with the experiments, it also showed that Intel's EDA tool Quartus Prime is still lacking proper software defenses to ensure confidentiality and integrity of IP files. We continued to extend the template project by a timer to measure request latencies and to issue different type of request.
After having everything set up, we determined that the request latencies clearly differ depending on the state a cache line has in the host. We found that distinguishing between clean and dirty cache lines is possible, however it remains an open problem where the additional delay for clean lines originates from. Furthermore, we found that requests via CXL.cache have direct access to the host LLC and can access all ways in a cache set. This was previously not possible with traditional direct cache access technologies like Intel DDIO. However, because CXL.cache behaves otherwise similar to the documented behavior of DDIO, we infer that the host likely uses a mechanism derived from DDIO to

implement the direct cache access. We also hypothesized that an attacker can use incoming snoop requests of the host to monitor a victims access patterns.
Most importantly, we find that there are no isolation mechanisms to limit the access of a CXL device to a certain section of host memory. Once the system is configured to allow CXL.cache requests to host memory, every CXL.cache-capable device will be able to access the entire host memory. Thus, CXL.cache in its current form is not suitable for a multi-tenancy cloud scenario, making our initially assumed attacker model unrealistic. This is because attackers with unrestricted access to CXL.cache do not need to utilize complicated cache attacks to exfiltrate sensitive data, they already have full access to physical memory. Due to that we started exploring Intel TDX since the threat model of Trusted Execution Environments (TEEs) includes a malicious hypervisor and malicious I/O devices. However, due to time constraints and our unfamiliarity of working with TDX, we could only perform a naive test which found that CXL.cache read requests do not seem to have a different view on physical memory while a VM protected by TDX is running. Thus, we conclude that a CXL device can not trivially read the encrypted memory of a protected VM. However, a more thorough analysis is needed.

Glossary

AFU Accelerator Functional Unit 11, 15, 16, 19, 21

AI artificial intelligence 1

ATE Atomic Test Engine 19

ATS Address Translation Services are an extension of PCIe which allow I/O devices to request virtual to physical address translation on their own. 7, 25

AXI It is an bus protocol typically used for on-chip communication and was invented by ARM as part of their *Advanced Microcontroller Bus Architecture*. There are multiple revisions and variants of it. AXI is widely adopted and the most commonly used protocol to interconnect multiple IP blocks in Xilinx FPGAs, however is it also commonly used in Intel FPGAs. The version used as an interface to the CXL IP is AXI4-MM. The latest revision is AXI5 [ARM18]. 19–23

CCI-P Core Cache Interface Protocol 21, 22

CPU central processing unit 1, 2, 5–7, 11, 26, 30, 33, 36, 40

CXL Compute Express Link iii, v, 1–3, 5–8, 11, 13–16, 19–23, 25–37, 39

D2H device to host 6, 7, 15, 16, 20, 26, 27

DDIO Intel Data Direct I/O Technology enables I/O devices to directly access the LLC on the host. Thus allowing for much lower latency as unnecessary accesses to main memory can be skipped. 2, 10, 11, 29–31, 36

DDR Double Data Rate 6

DMA Direct Memory Access 6, 10, 30, 31

ECC error correction code 36

EDA Electronic design automation, a collective term for software tools used to design electronic systems like circuits or printed circuit boards. In this thesis we use the EDA tool Quartus Prime, which is published by Altera in order to program their FPGAs 2, 3, 17, 36, 39

Example Design Terminology found in the Intel / Altera ecosystem. It is a hardware design provided by the designer of an IP to showcase an example on how to integrate the IP block in larger designs. 11, 14–16, 19–23, 27, 41

FPGA field programmable gate array iii, v, 1, 3, 5, 11, 13, 14, 21, 22, 36, 39, 40

GPA guest physical address 33, 36

GPU graphics processing unit 1

H2D host to device 6, 7, 23, 27

HDM host-managed device memory 20

HPA host physical address 7, 25, 33, 36

HPC hardware performance counter 26, 27

I/O Input-Output 2, 7, 10, 11, 13, 25, 26, 33, 36, 37, 39, 40

IOMMU Input-Output Memory Management Unit iii, v, 2, 25

IP Short for Intellectual Property. When creating an FPGA design, one often does not implement all hardware components from scratch and rather uses IPs, which are reusable hardware blocks developed by others. 2, 3, 11, 13–17, 19–23, 26, 27, 29, 31, 35, 36, 39–41

LLC The last level cache in a CPU cache hierarchy. iii, v, 2, 8–11, 13, 28–31, 33, 36, 39

LLM large language model 16, 17, 36

MESI A snooping-based cache coherency protocol using the states Modified, Exclusive, Shared and Invalid. 5, 7, 27

MESIF Adapted version of the MESI protocol with additional *Forward* state. 5

MMIO Memory-mapped I/O 13, 19

MOESI Adapted version of the MESI protocol with additional *Owned* state. 5

MSR model specific register 10, 31

NDA non-disclosure agreement 15

NIC network interface card 10

OPAE Open Programmable Acceleration Engine 11

OS operating system 13

PCIe Peripheral Component Interconnect Express 6, 7, 10, 13, 19, 21, 29, 30, 32, 39

PMP physical memory protection 36

RCiEP Root Complex Integrated Endpoint 13

SCA side-channel attack 1, 5

SEAM Secure Arbitration Mode 33

TD Trusted Domain 33, 34, 36

TDX Trust Domain Extensions iii, v, 33, 34, 36, 37

TEE Trusted Execution Environment 37

UEFI unified extensible firmware interface 13, 32

UPI Ultra Path Interconnect 6, 21

User Guide Terminology found in the Intel / Altera ecosystem. This is a documentation tightly coupled to an IP and its corresponding Example Design. Among other things, these guides covers the interfaces to the IP, showcase block diagrams of the IP architecture, and register address space 14, 15

VM virtual machine 33, 37

References

[Ali25] Alibaba Cloud. Instance Types of Elastic Compute Services. https://www.alibabacloud.com/help/en/ecs/user-guide/instance-specification-naming-and-classification, 2025. Accessed: 2025-11-20.

[Alt25a] Altera. Agilex 7 FPGAs and SoCs Device Overview. https://web.archive.org/web/20251101170930/https://cdrdv2-public.intel.com/666707/ag-overview-683458-666707.pdf, 2025. Accessed: 2025-11-01.

[Alt25b] Altera. Community Forum. https://web.archive.org/web/20251208191025/https://community.altera.com/discussions/fpga-device/agilex-7-i-series-dev-kit-pipe-direct-for-custom-pciecxl-controller/324180/replies/324181, 2025. Accessed: 2025-12-08.

[Ama25] Amazon. FPGA powered AWS-EC2-F2 instances. https://web.archive.org/web/20251208191411/https://aws.amazon.com/de/ec2/instance-types/f2/, 2025. Accessed: 2025-12-08.

[AMD20] AMD. Infinity Fabric. https://web.archive.org/web/20241230132816/https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/other/56978.pdf, 2020. Accessed: 2025-11-22.

[Ant25] Anthropic. Claude Code. https://web.archive.org/web/20251124113723/https://www.claude.com/product/claude-code, 2025. Accessed: 2025-11-24.

[ARM18] ARM. AMBA 5. https://web.archive.org/web/20251111130528/https://www.arm.com/architecture/system-architectures/amba/amba-5, 2018. Accessed: 2025-11-11.

[BIME18] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. CacheShield: Detecting cache attacks through self-observation. In

Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18, page 224–235, New York, NY, USA, 2018. Association for Computing Machinery.

[BPSS24] Pavitra Bhade, Joseph Paturel, Olivier Sentieys, and Sharad Sinha. Lightweight hardware-based cache side-channel attack detection for edge devices (edge-cascade). *ACM Trans. Embed. Comput. Syst.*, 23(4), June 2024.

[CFQP23] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. Fight hardware with hardware: Systemwide detection and mitigation of side-channel attacks using performance counters. *Digital Threats*, 4(1), March 2023.

[COV+24] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *ACM Comput. Surv.*, 56(9):238:1–238:33, 2024.

[CXL19] CXL Consortium. Compute Express Link Specification Revision: 1.1. `https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.1-Specification.pdf`, 2019. Accessed: 2025-06-24.

[DSBB24] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Computing Surveys*, 56(11):1–37, November 2024.

[FLC+25] Xiaoli Fang, Xuhui Liu, Chun-Zhang Chen, Liang Wang, Quan Pan, and Hanming Wu. Custom Design of CXL Controller on Intel FPGA R-Tile. In *2025 Conference of Science and Technology of Integrated Circuits (CSTIC)*, pages 1–3, 2025.

[HP18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6 edition, 2018.

[Int12] Intel. Intel Data Direct I/O Technology: A Primer. `https://web.archive.org/web/20250824100144/https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf`, 2012. Accessed: 2025-12-08.

[Int19] Intel. Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P). `https://web.archive.org/web/`

20250916104935/https://www.intel.com/content/www/us/en/docs/programmable/683193/current/introduction.html, 2019. Accessed: 2025-11-18.

[Int22] Intel. Xeon Processor Technical Overview. https://web.archive.org/web/20251122175922/https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, 2022. Accessed: 2025-11-22.

[Int23] Intel. Product Brief, 5th Gen Intel Xeon Processors. https://web.archive.org/web/20251101153501/https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2023-11/5th-gen-xeon-processors-product-brief.pdf, 2023. Accessed: 2025-11-01.

[Int24] Intel. Intel Knowledge Base. https://web.archive.org/web/20251027161912/https://www.intel.com/content/www/us/en/support/articles/000027820/processors/intel-xeon-processors.html, 2024. Accessed: 2025-10-27.

[Int25a] Intel. TDX Module Architecture Application Binary Interface. https://web.archive.org/web/20251208041354/https://www.intel.de/content/www/de/de/content-details/865802/intel-tdx-module-abi-specification.html, 2025. Accessed: 2025-12-07.

[Int25b] Intel. Trust Domain Extensions. https://web.archive.org/web/20250823124914/https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/trust-domain-extensions.html, 2025. Accessed: 2025-11-27.

[KFC+24] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. SoK: Can we really detect cache side-channel attacks by monitoring performance counters? In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 172–185, New York, NY, USA, 2024. Association for Computing Machinery.

[KGA+20] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Net-

work. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 20–38. IEEE, 2020.

[MAPMK+20] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A. Jiménez. Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1124–1137, 2020.

[Mic25a] Micron. CXL-based memory. `https://www.micron.com/products/memory/cxl-memory`, 2025. Accessed: 2025-11-23.

[Mic25b] Microsoft. FPGA powered Microsoft Azure NP instances. `https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/fpga-accelerated/np-series?tabs=sizebasic`, 2025. Accessed: 2025-06-10.

[Nvi14] Nvidia. NVLink Whitepaper. `https://web.archive.org/web/20250221153247/http://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf`, 2014. Accessed: 2025-11-22.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[PS19] PCI-SIG. *PCI Express Base Specification, Revision 5.0*, May 2019.

[PTV22] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Double trouble: Combined heterogeneous attacks on Non-Inclusive cache hierarchies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3647–3664, Boston, MA, August 2022. USENIX Association.

[RFKG25] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A systematic evaluation of novel and existing cache side channels. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[RIS25] RISC-V International. RISC-V Instruction Set Manual: Privileged Architecture. `https://web.archive.org/web/20251208030628/https:`

//docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf, 2025. Accessed: 2025-12-06.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212. ACM, 2009.

[Sam25] Samsung. CXL Memory. https://semiconductor.samsung.com/cxl-memory/, 2025. Accessed: 2025-11-23.

[SKLG22] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[SSE+22] Julian Speith, Florian Schweins, Maik Ender, Marc Fyrbiak, Alexander May, and Christof Paar. How not to protect your IP - an industry-wide break of IEEE 1735 implementations. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1656–1671. IEEE, 2022.

[Ter23] Terasic. Mercury A2700 Accelerator Card. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1300, 2023. Accessed: 2025-11-23.

[VKM19] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 39–54. IEEE, 2019.

[WMYZ22] Minjun Wu, Stephen McCamant, Pen-Chung Yew, and Antonia Zhai. Predator: A cache side-channel attack detector based on precise event monitoring. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 25–36, 2022.

[WTM+20] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on heterogeneous FPGA-CPU platforms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):169–195, 2020.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association.

[YSG+19] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 888–904. IEEE, 2019.